Assignment 4

The goal of this assignment is to create a C program which implements a singly linked list. You will implement this program in a class file named LinkedList.c and a corresponding header file named LinkedList.h . This header file will contain all includes, constants, and function prototypes you define. You will also be provided the Givens.h, Testing.h, and Testing.c files. LinkedList.h will include Givens.h and Testing.h will include LinkedList.h .

Givens.h

Givens.h is provided with this assignment. Givens.h defines a Node struct with two member fields: data and next. The data field is of type char* and stores a pointer to a character array (i.e. a String). The next field is of type struct Node* and stores a pointer to the next node in the list. This struct will be used as the foundation of our linked list.

LinkedList.c

LinkedList.c will contain all functions necessary to create, access, and modify a singly linked list. LinkedList.c must include LinkedList.h and LinkedList.h must include Givens.h . This will allow LinkedList.c to use the Node struct defined in Givens.h .

LinkedList.c will define the following linked list functions:

```
struct Node* createNode(char* data);
```

The createNode function will take in a character array and return a pointer to a newly initialized Node struct. To create a new Node struct, createNode will call malloc to dynamically allocate the necessary memory. malloc will return a void* which must be cast to a struct Node* and stored. Once allocated, the member fields of the new Node struct must be initialized.

NOTE: createNode is working with a Node *pointer*, not a Node struct directly. While the dot operator is used to access the member fields of a struct, we have a pointer to a struct. So, we need to dereference the pointer before accessing any of its member fields. While we could combine the dereference operator and the dot operator to accomplish this, C provides a structure dereference operator for this exact purpose. Its use is highly recommended.

The next field stores a pointer to the next node in the list. Because we're only creating a single Node, next will be initialized to NULL. The data field stores a pointer to a character array and should be initialized to a char*.

NOTE: While createNode is given a char* as a parameter and the data field stores a char*, we **cannot** simply copy the value in the data parameter into the new Node's data field. The new Node **must** be guaranteed ownership of the character array it's referencing in order to prevent its unwanted modification. For example, if the character array pointed to by the new Node is also pointed to by other pointers in the program, then other elements of the program could modify the Node's data without accessing the Node directly. This is not the expected behavior of any data structure!

To guarantee the new Node has ownership of its character array, createNode will call malloc to dynamically allocate memory for a new character array. It will then copy the characters from the character array pointed to by the data parameter into the newly allocated character array. Lastly, a pointer to the newly allocated character array is stored in the Node's data field.

```
void insertAtEnd(struct Node** head, struct Node* newNode);
```

The insertAtEnd function will take in the head of a linked list and a Node struct. It will append the given Node to the end of the given linked list.

NOTE: The head parameter is of type struct Node**. Meaning head stores a pointer to a pointer to a Node struct. Why? Well, say a caller function passed insertAtEnd a pointer to a Node struct. If insertAtEnd needed to alter which Node struct was the head of the list, it would have no way of passing this information back to the caller. The caller would have a pointer to the old head Node struct and from the caller's perspective, the list would be unchanged. By making insertAtEnd's head parameter a pointer to a pointer to a Node struct, a caller function must pass a pointer to the storage location of the head Node struct pointer. This allows the head of the list to be modified. If insertAtEnd must alter which Node struct is the head of the list, it dereferences the head parameter a single time to access the storage location where the *caller* stores the head Node struct pointer. Modifying the pointer at that storage location will change the head node for both insertAtEnd *and* the caller function. This rational applies for all LinkedList.c functions with parameters of type struct Node**

If head points to a storage location containing NULL, then the newNode pointer should be stored at that storage location. If head points to a storage location containing a Node pointer, then the newNode pointer must be stored in the next field of the last Node in that list. If the head parameter itself is NULL, no action should be taken and no runtime errors should occur.

```
struct Node* createList(FILE* inf);
```

The createList function takes in a pointer to an opened file. createList will read the given file line by line, storing each line of the file into a Node struct. The first Node struct created will be the head of the new linked list and all subsequently created Node structs must be linked to the

previously created Node struct. When the entire file is read createList will return a pointer to the head of a newly created linked list. createList is not responsible for closing the given file and should not do so. If createList is given an empty file, it should return NULL. The maximum size of a file line is provided as a constant in Givens.h .

NOTE: When reading a line of data from a file, many C functions do not strip out the trailing \n character. You must remove this \n from your character array before storing it in a Node struct. This can be accomplished with the following statement (shamelessly stolen from here):

```
fileLine[strcspn(fileLine, "\n")] = 0;
```

```
struct Node* removeNode(struct Node** head, int index);
```

The removeNode function will take in the head of a linked list and the index of the Node struct to be removed from the linked list. removeNode will traverse the given linked list until the given index is reached. It will then remove the Node struct at that index from the linked list. A pointer to the removed Node struct will be returned.

removeNode should return NULL for the following edge cases:

- If head points to a storage location containing NULL
- If the head parameter itself is NULL
- If the given index is out of bounds
- If the given index is negative

```
void traverse(struct Node* head);
```

The traverse function will take in the head of a linked list and traverse over every Node struct in the given list. For each node in the list, traverse will use printf to print the value of the data field followed by a \n character. When the data field of every node in the list has been printed, traverse will terminate.

```
void freeNode(struct Node* aNode);
```

The freeNode function will take in a pointer to a Node struct and free it's memory, thus deallocating the struct and allowing the memory to be used for other purposes. freeNode must free all allocated memory associated with a Node struct. This includes the Node struct itself and any memory allocated for that specific nodes use.

NOTE: You can only free memory allocated by malloc or one of its variants. Statically allocated memory cannot be explicitly freed and attempting to do so will cause a runtime error.

```
void freeList(struct Node** head);
```

The freeList function will take in the head of a linked list and will free the memory for all nodes in the list, including the head node itself. Thus, deallocating the entire linked list. This should result in the storage located pointed to by head being NULL when freeList concludes.

Testing.c

Testing.c is provided with this assignment. Testing.c implements an int main function which includes code to prompt a user to input a filepath at the command line until a valid filepath is input. Once a valid filepath is provided, int main will open the file for reading and store a pointer to the open file in the `infile` variable. While the contents of Testing.c will not be graded, you can use the main function in Testing.c to, well, test the linked list functions implemented in LinkedList.c . I cannot stress enough how vital it is to do basic testing in Testing.c as Gradescope is *grading* software not *testing* software.

NOTE: You **cannot** define an int main function in LinkedList.c, doing so will break Gradescope.

Final Notes

A set of text files will be provided for testing. There will be 4 text files, all containing a different amount of input data. These files are the same files Gradescope will use for testing. Unfortunately, the tests themselves are not portable and cannot be provided directly.

There is no recommended development environment for this assignment and you are welcome to use whatever application you are most comfortable with.

Grading Rubric:

Category	Points
Header files correctly utilized / included	10
Functions and Classes properly defined	10
LinkedList.c correctly implemented	60
Code is well commented	10
Code is well formatted	10
Total	100